Cutting Edge OpenGL Techniques

Session 212

Cutting Edge OpenGL Techniques

Alex Vlachos
ATI 3D Application Research Group
Lead Programmer, Demo Team
avlachos@ati.com

Marwan Y. Ansari
ATI 3D Application Research Group
mansari@ati.com

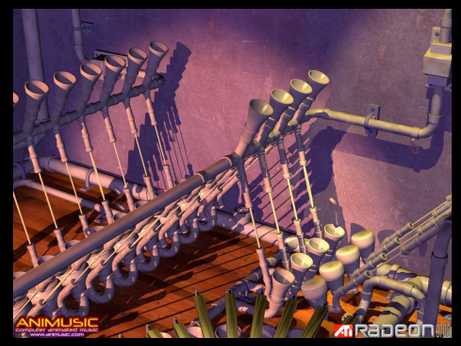
Rav Dhiraj
ATI Mac Research Team
rdhiraj@ati.com

Outline

- Normal Mapper Tool—Adding Detail to the Car Paint Demo

- Video and Post-Processing Shader Effects

- Shadow Techniques Used In RADEON 9700 Animusic Demo



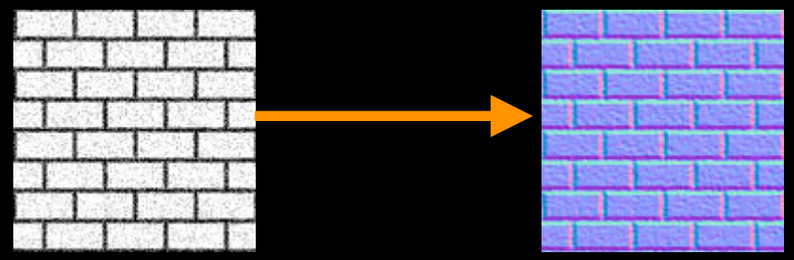
Normal Mapper Tool

Developed by Dave Gosselin, ATI

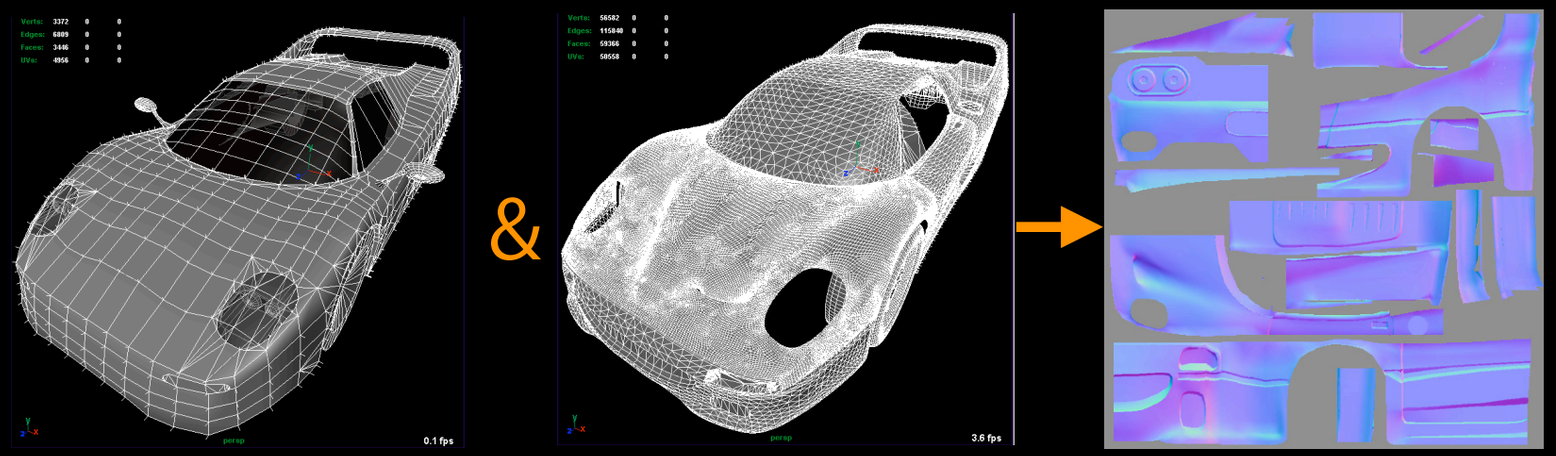
Demo

Normal Maps

Commonly created from a height map:

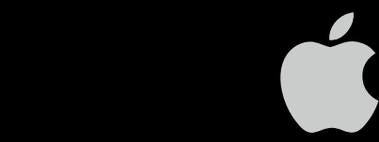


Using low and high resolution models is becoming widely adopted:



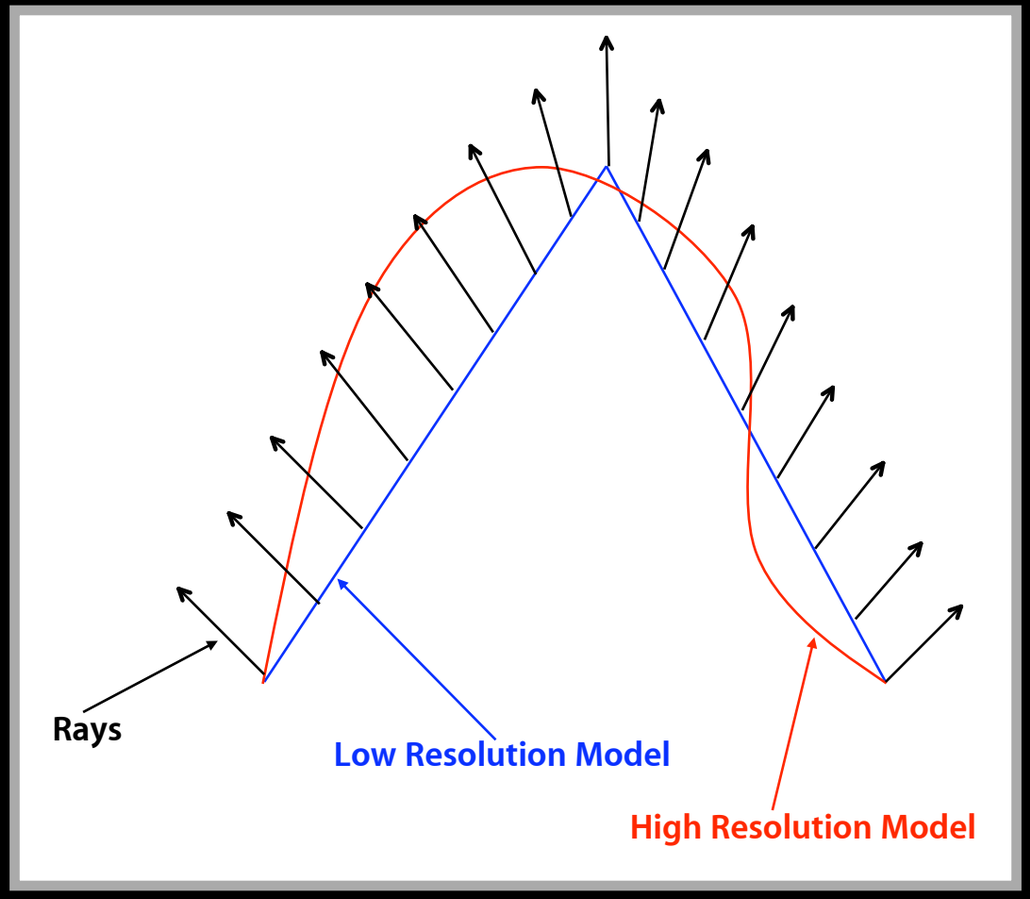
ATI's Normal Mapper Tool

- Command line interface that generates a bump map from a high and low resolution model (and a height map for fine details)
- High resolution model requires vertices and normals
- Low resolution model requires vertices, normals, and one set of texture coordinates
- The output normal map is applied to the low resolution model at runtime



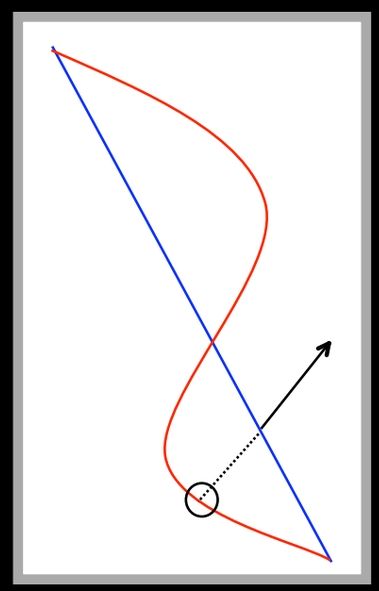
Ray Casting

- For each texel mapped onto the low resolution model, a ray is cast out from the low resolution model and intersected with the high resolution model (Multiple rays if super sampling)



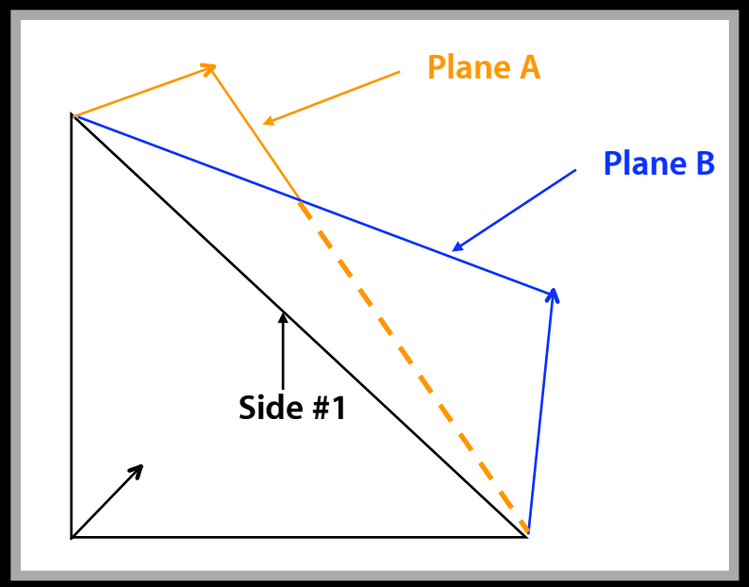
Ray Casting (Cont.)

- If multiple polygons on the high resolution model are intersected, the one closest to the origin of the ray is chosen
- Super Sampling: RADEON 9700 Car Paint Demo was generated with 36 rays per texel
- Negative intersections are valid:



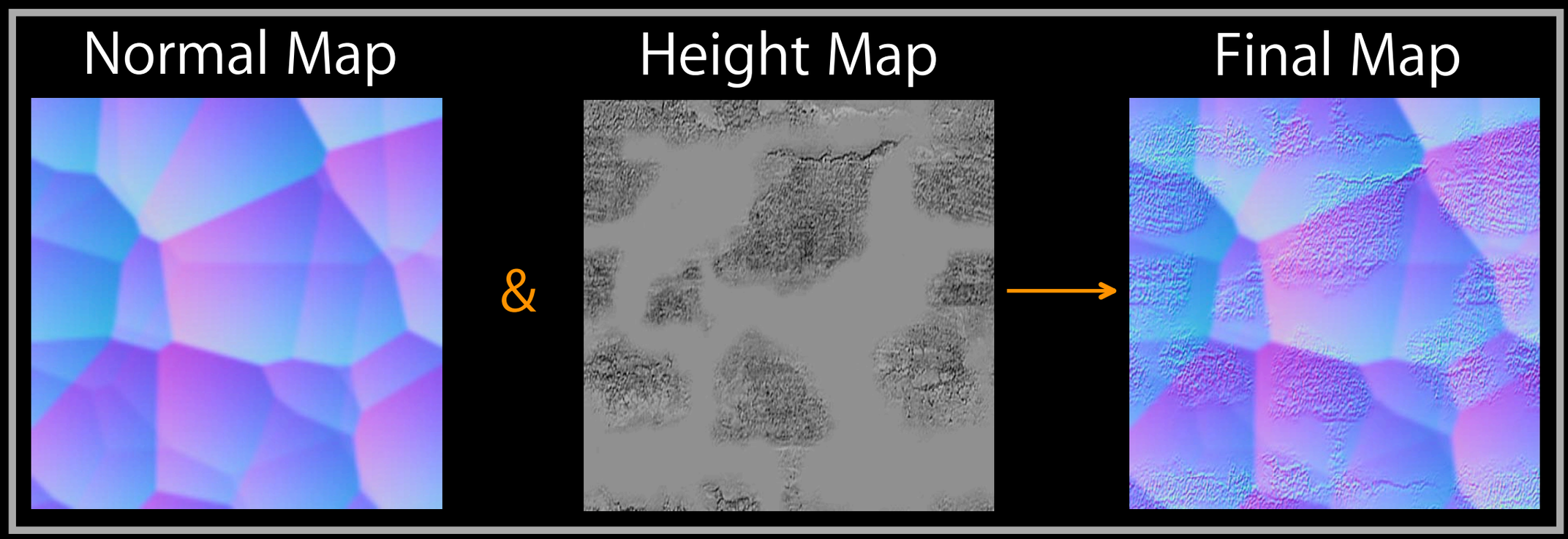
Beam Optimization

- As a preprocess, to minimize the number of intersection tests for each ray, a list of polygons on the high resolution model is generated for each polygon on the low resolution model
- A 3-sided beam consisting of 6 planes is used. Each side of the beam is represented by 2 planes. If the plane test passes for either of the 2 planes, the test for that side of the beam passes.



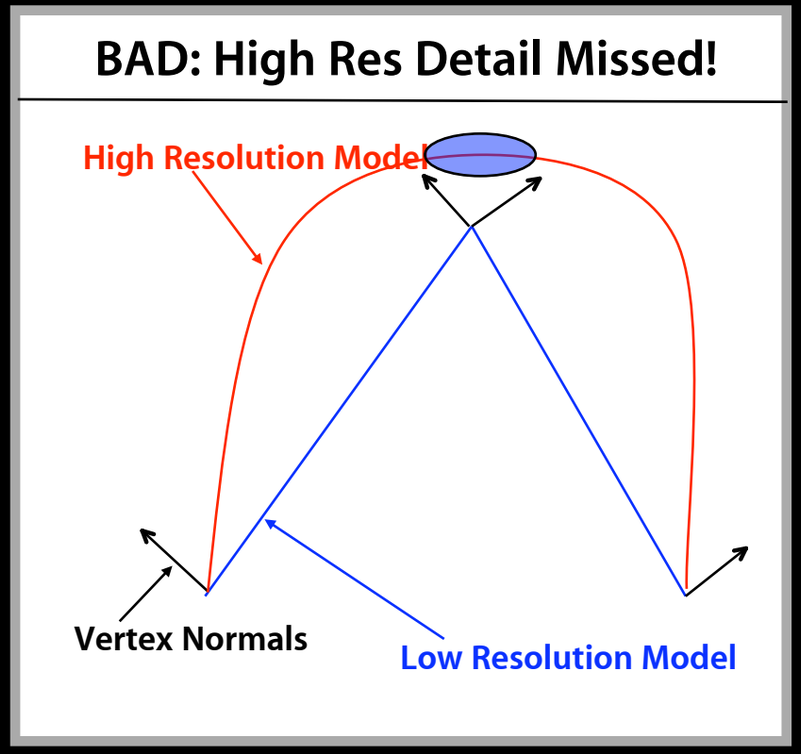
Combining With A Height Map

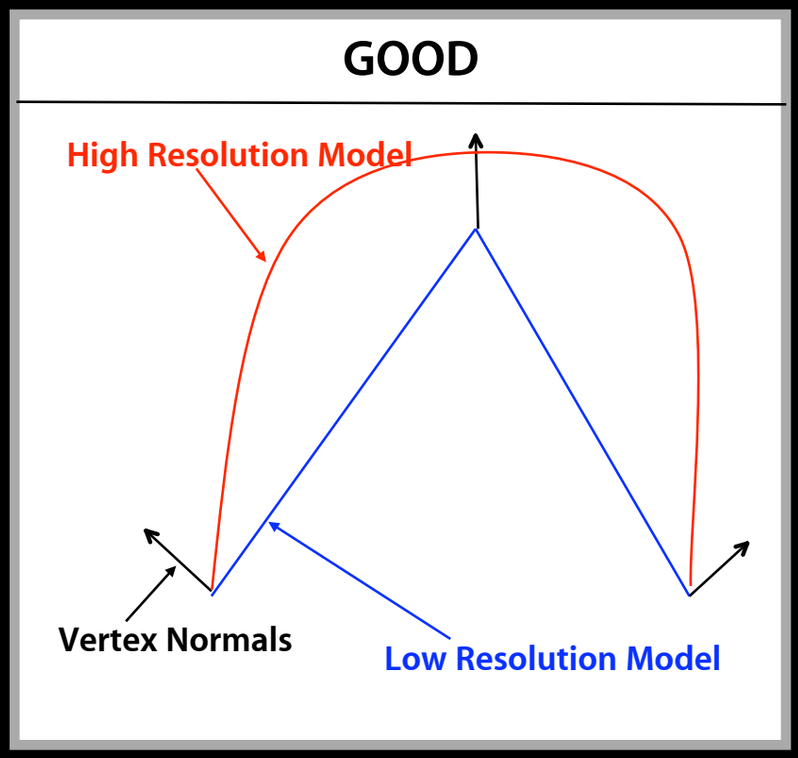
- A height map can be combined with the ray cast normal map to provide micro detail to the normal map
- Scars on skin, imperfections in rock, bumps in bricks, etc.



Authoring Normals For Low Resolution Model

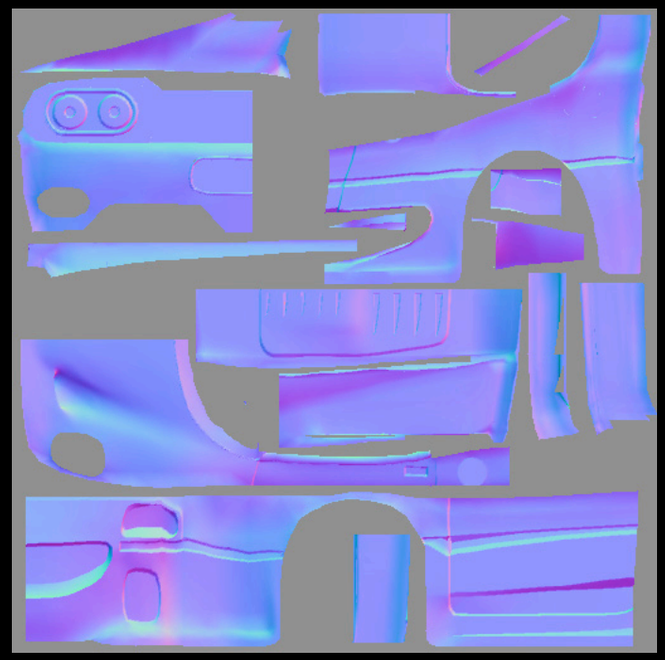
- Artists must make sure that rays cast based on the normals from the low resolution model can accurately represent the high resolution model





Spacing Texture Coordinates

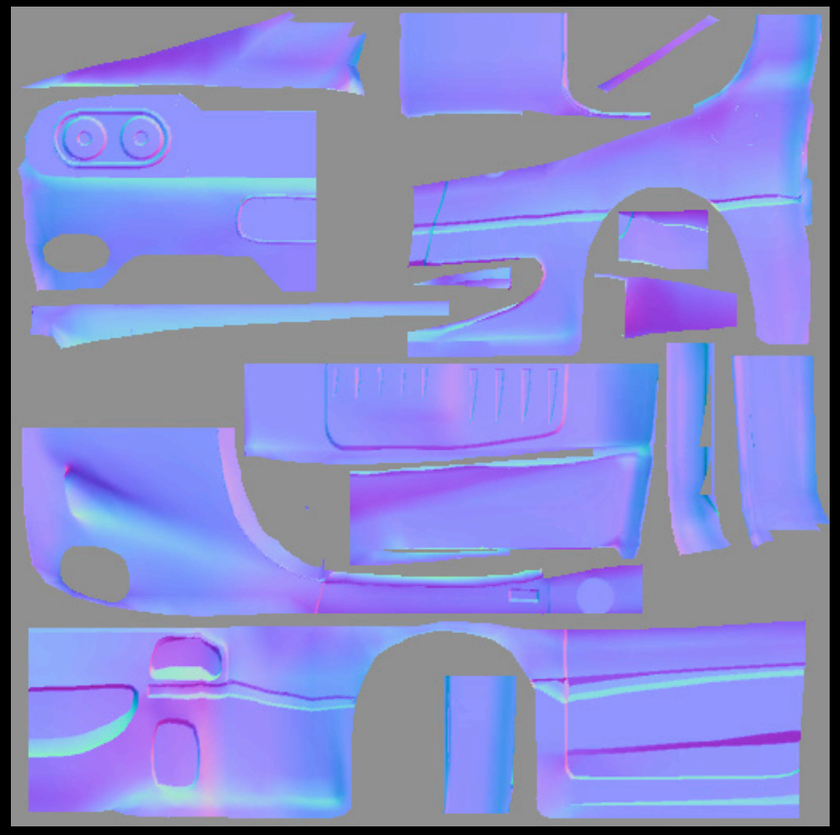
- Space must be left between surfaces so there is room for a dilation filter
- Bilinear fetches and mip map generation requires this



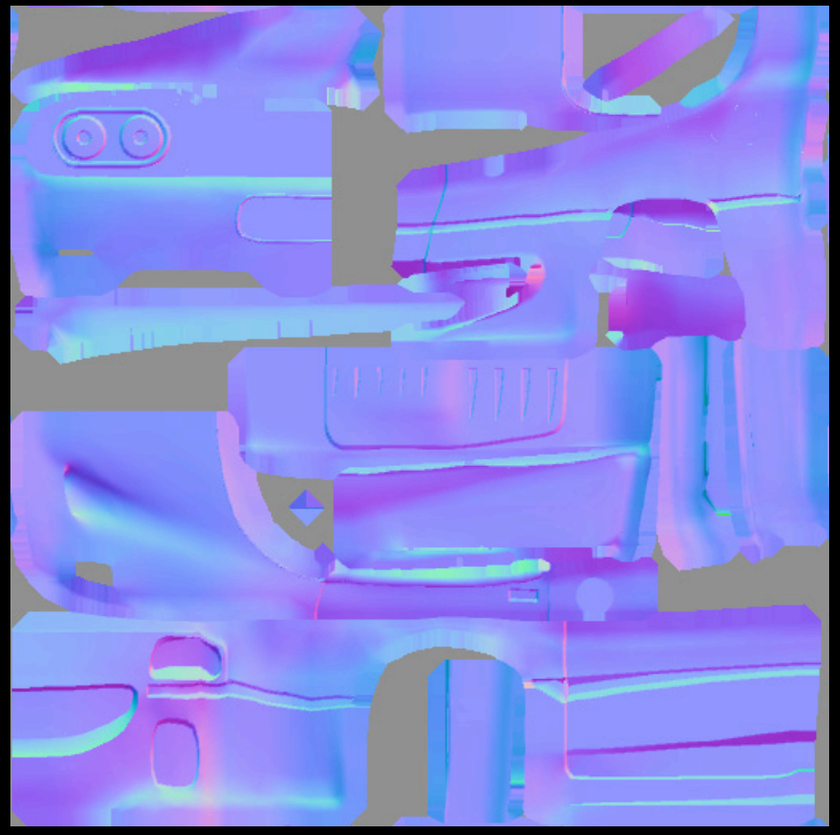
Normal Map Dilation Filter for Bilinear Fetches

- A 4-tap filter is used to grow the normals to neighboring pixels. This is necessary for generating mip maps and bilinear fetches. ATI's tool runs this filter four times.

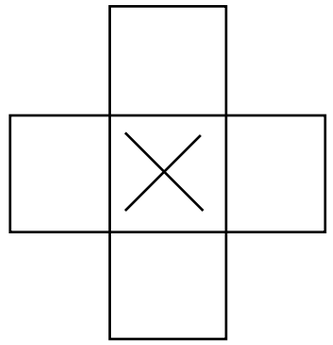
Before Filter



After Filter



Dilation Filter



Tangent Space

- Tangent space in the normal map computations must match the tangent space of the run-time model
- If normal map is generated in world space, tangent space isn’t needed, unless model is skinned

Normal Map Summary

- Defined how a low and high resolution model are used to generate a normal map
- Beam optimization
- Combining with a height map
- Artist notes:
  - Watch for unshared vertex normals!
  - Leave room in textures for grow kernels!

Video and Post-Processing Shader Effects

- Optimized Sepia Tone
- Sobel Edge Detection
- Posterized NPR Rendering
- Real-Time FFT

Sepia Tone

Demo

Sepia Tone

- Looks like antique photograph
- Doesn’t require a lookup table
- Doesn’t require human interaction
- Executed through a color space conversion of RGB → YIQ → RGB
- Brute-force color conversion is just a matrix multiply
  - We’ll take some shortcuts…

Sepia Tone Shader Algorithm

1. Compute Y component of YIQ from the RGB pixel color (Just a dp3)
2. Set the I component of YIQ to 0.2
3. Set the Q component of YIQ to 0.0
4. Convert YIQ back to RGB color space

- Since YIQ is set to (Y, 0.2, 0.0), we can optimize the color space conversion!
- Normally, it's a 3x3 matrix multiply; Now it's just an add since I and Q are constants!

Sepia Tone Fragment Shader

```
!!ARBfp1.0
# Matrix to move from RGB to YIQ as per Foley, van Dam et. al.
# Note: c1, c2, c3 aren't actually used since we assume I=0.2 and Q=0.0
PARAM  c0 = { 0.299,  0.587,  0.114, 0.000 };
PARAM  c1 = { 0.596, -0.275, -0.321, 0.000 };
PARAM  c2 = { 0.212, -0.523,  0.311, 0.000 };
PARAM  c3 = { 0.700,  0.500,  0.250, 0.125 };

# Vector to move from YIQ(Y, 0.2, 0.0) to RGB
PARAM  c4 = { 0.1912, -0.0544, -0.2216, 0.000 };

ATTRIB  t0 = fragment.texcoord[0];
TEMP  r0, r1;
OUTPUT  oC0 = result.color;

# Sample texture
TEX r0, t0, texture[0], 2D;

# Calculate Y only since I and Q are hard-coded
DP3 r1.x, r0, c0;

# Convert from YIQ color space back to RGB (Just an add)
ADD r1, r1.x, c4;

# Result is in Sepia tone
MOV oC0, r1;

END
```

Sobel Edge Detection

Demo

Sobel Edge Detection

- Gradient filter
  - Compute Horizontal Gradient (δu)
  - Compute Vertical Gradient (δv)
- Result = sqrt(δu^2 + δv^2)
  - Or Result = abs(δu) + abs(δv)

Horizontal Kernel

$\delta u =$

-1	-2	-1
0	0	0
1	2	1

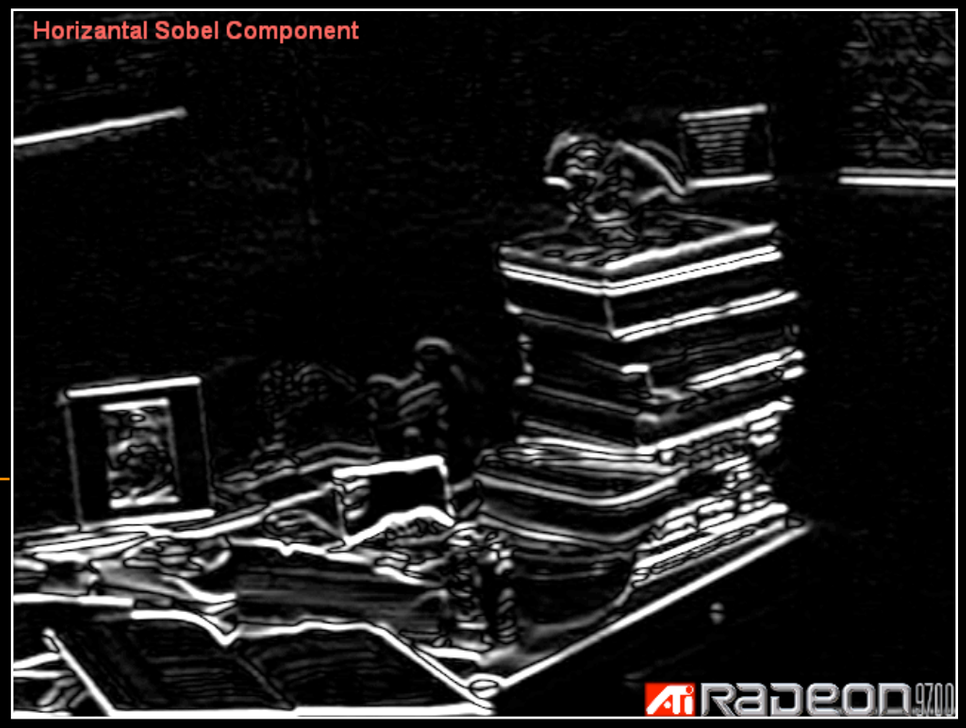
Vertical Kernel

$\delta v =$

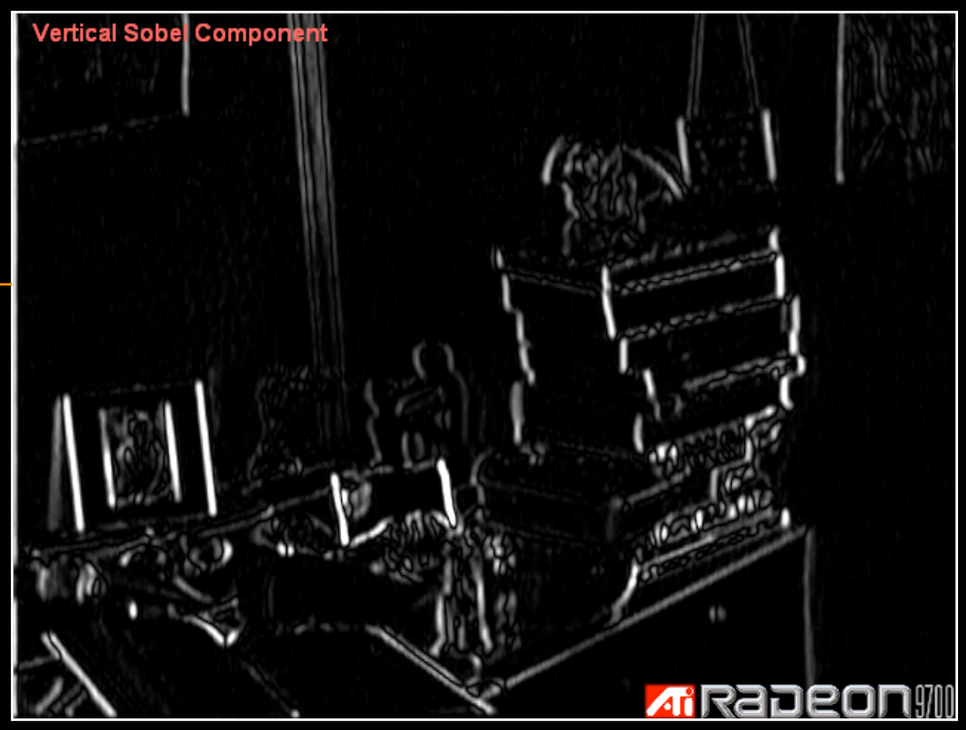
-1	0	1
-2	0	2
1	0	1

Sobel Partial Output

Horizontal Result



Vertical Result



Sobel Code

```
!!ARBfp1.0
TEMP r0, r1, r2, r3, r4, r5, r6, r7, r8;
TEMP r9, r10, r11;
ATTRIB t0 = fragment.texcoord[0];
ATTRIB t1 = fragment.texcoord[1];
OUTPUT oC0 = result.color;

PARAM c0 = {0.33, 0.33,  0.33, 0.33};
PARAM c1 = {2.00, 2.00,  2.00, 2.00};
PARAM offset[8]  = { program.env[0..7] };

# Sample center
TEX r0, t0, texture[0], 2D;

# Generate coords and sample surrounding pixels
ADD r1.xy, t0, offset[0];  # top left
ADD r2.xy, t0, offset[1];  # top center
ADD r3.xy, t0, offset[1];  # top right
ADD r4.xy, t0, offset[2];  # left
ADD r5.xy, t0, offset[3];  # right
ADD r6.xy, t0, offset[4];  # bottom left
ADD r7.xy, t0, offset[5];  # bottom center
ADD r8.xy, t0, offset[6];  # bottom right

TEX r1, r1, texture[0], 2D;
TEX r2, r2, texture[0], 2D;
TEX r3, r3, texture[0], 2D;
TEX r4, r4, texture[0], 2D;
TEX r5, r5, texture[0], 2D;
TEX r6, r6, texture[0], 2D;
TEX r7, r7, texture[0], 2D;
TEX r8, r8, texture[0], 2D;

# Convert to B&W
DP3 r0, r0, c0;
DP3 r1, r1, c0;
DP3 r2, r2, c0;
DP3 r3, r3, c0;
DP3 r4, r4, c0;
DP3 r5, r5, c0;
DP3 r6, r6, c0;
DP3 r7, r7, c0;
DP3 r8, r8, c0;

# VERTICAL Sobel
ADD r9, r1, r6;
MUL r10, r4, c1;
ADD r11, -r9, -r10;
ADD r9, r3, r8;
MUL r10, r5, c1;
ADD r9, r9, r10;
ADD r11, r9, r11;

# HORIZONTAL Sobel
MOV r4, r0;
ADD r9, r1, r3;
MUL r10, r2, c1;
ADD r0, -r9, -r10;
ADD r9, r6, r8;
MUL r10, r7, c1;
ADD r9, r9, r10;

# Combine Sobels
ABS r9, r9
ABS r11, r11
ADD oC0, r9, r11;

END
```

Hand-Drawn Posterized Style

- Inspired by Waking Life
- Apply edge-preserving Kuwahara filter
- Optionally composite edges on top

Frame from *Waking Life*

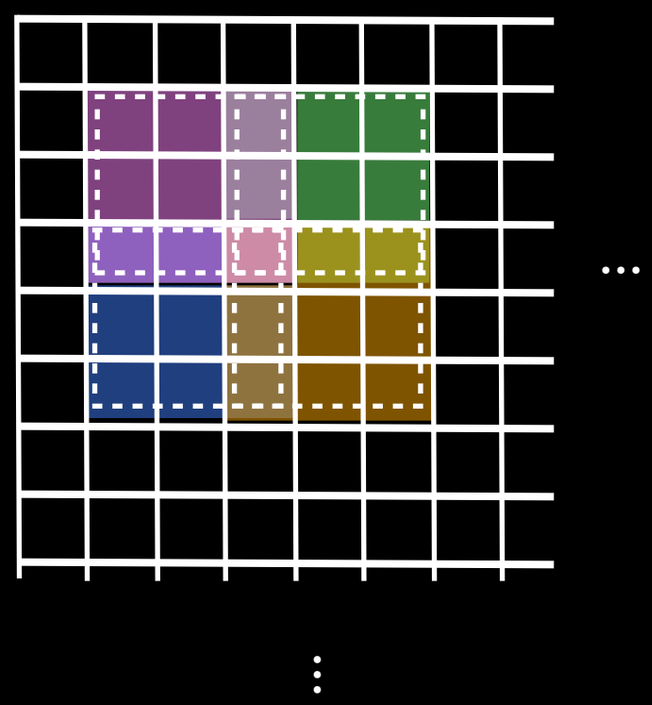
Demo

5x5 Posterization Output

Posterization

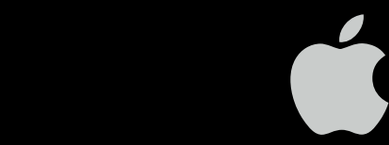
- Kuwahara Filter (Theory)
  - Non-linear edge-preserving smoothing operation
  - 4 overlapping 3x3 regions
  - Compute variance and mean color for each region

  - Output is the mean color of the region with smallest variance



5x5 Kuwahara Implementation

- Apply Kuwahara filter
  - Compute the Mean & Variance for each of the four regions
    - 3x3 Kernel
    - Store Mean in RGB
    - Store Variance in Alpha
  - Sample the Mean and Variance Pair keeping the pair with the lowest variance
  - Total of 2 Passes for a single Kuwahara Filter
- Add Edge Detection pass for outlines—Sobel edge detection filter from earlier slides



Mean and Variance Computation

```
!!ARBfp1.0
PARAM  c0 = { 0.111111, 0, 0, 0 };
ATTRIB t0 = fragment.texcoord[0];
TEMP r0, r1, r2, r3, r4, r5, r6, r7, r8, r10, r11;
OUTPUT oC0 = result.color;
PARAM  c[8] = { program.env[0..7] };

TEX r0, t0, texture[0], 2D;

# Calculate tex coords
ADD r1.xy, t0, c[0];
ADD r2.xy, t0, c[1];
ADD r3.xy, t0, c[2];
ADD r4.xy, t0, c[3];
ADD r5.xy, t0, c[4];
ADD r6.xy, t0, c[5];
ADD r7.xy, t0, c[6];
ADD r8.xy, t0, c[7];
TEX r1, r1, texture[0], 2D;
TEX r2, r2, texture[0], 2D;
TEX r3, r3, texture[0], 2D;
TEX r4, r4, texture[0], 2D;
TEX r5, r5, texture[0], 2D;
TEX r6, r6, texture[0], 2D;
TEX r7, r7, texture[0], 2D;
TEX r8, r8, texture[0], 2D;

# Sum the sample to compute the mean
ADD r10.xyz, r0, r1;
ADD r10.xyz, r10, r2;
ADD r10.xyz, r10, r3;
ADD r10.xyz, r10, r4;
ADD r10.xyz, r10, r5;
ADD r10.xyz, r10, r6;
ADD r10.xyz, r10, r7;
ADD r10.xyz, r10, r8;

# Compute the mean by dividing the total by
9
MUL r11.xyz, r10 , c0.x;

# Compute (mean-sample)
ADD r0.xyz, r11 ,-r0 ;
ADD r1.xyz, r11 ,-r1 ;
ADD r2.xyz, r11 ,-r2 ;
ADD r3.xyz, r11 ,-r3 ;
ADD r4.xyz, r11 ,-r4 ;
ADD r5.xyz, r11 ,-r5 ;
ADD r6.xyz, r11 ,-r6 ;
ADD r7.xyz, r11 ,-r7 ;
ADD r8.xyz, r11 ,-r8 ;

# Compute:  total += (mean-sample)^2
MUL r0.xyz, r0 , r0 ;
MAD r0.xyz, r1 , r1 , r0 ;
MAD r0.xyz, r2 , r2 , r0 ;
MAD r0.xyz, r3 , r3 , r0 ;
MAD r0.xyz, r4 , r4 , r0 ;
MAD r0.xyz, r5 , r5 , r0 ;
MAD r0.xyz, r6 , r6 , r0 ;
MAD r0.xyz, r7 , r7 , r0 ;
MAD r0.xyz, r8 , r8 , r0 ;

# Combine the variances of r, g, and b into
one value.
DP3 r11.w, r0 , c0.x;

# r11.xyz is the mean color of this region,
r11.w is the variance
MOV oC0 , r11;
END
```

Variance Selection

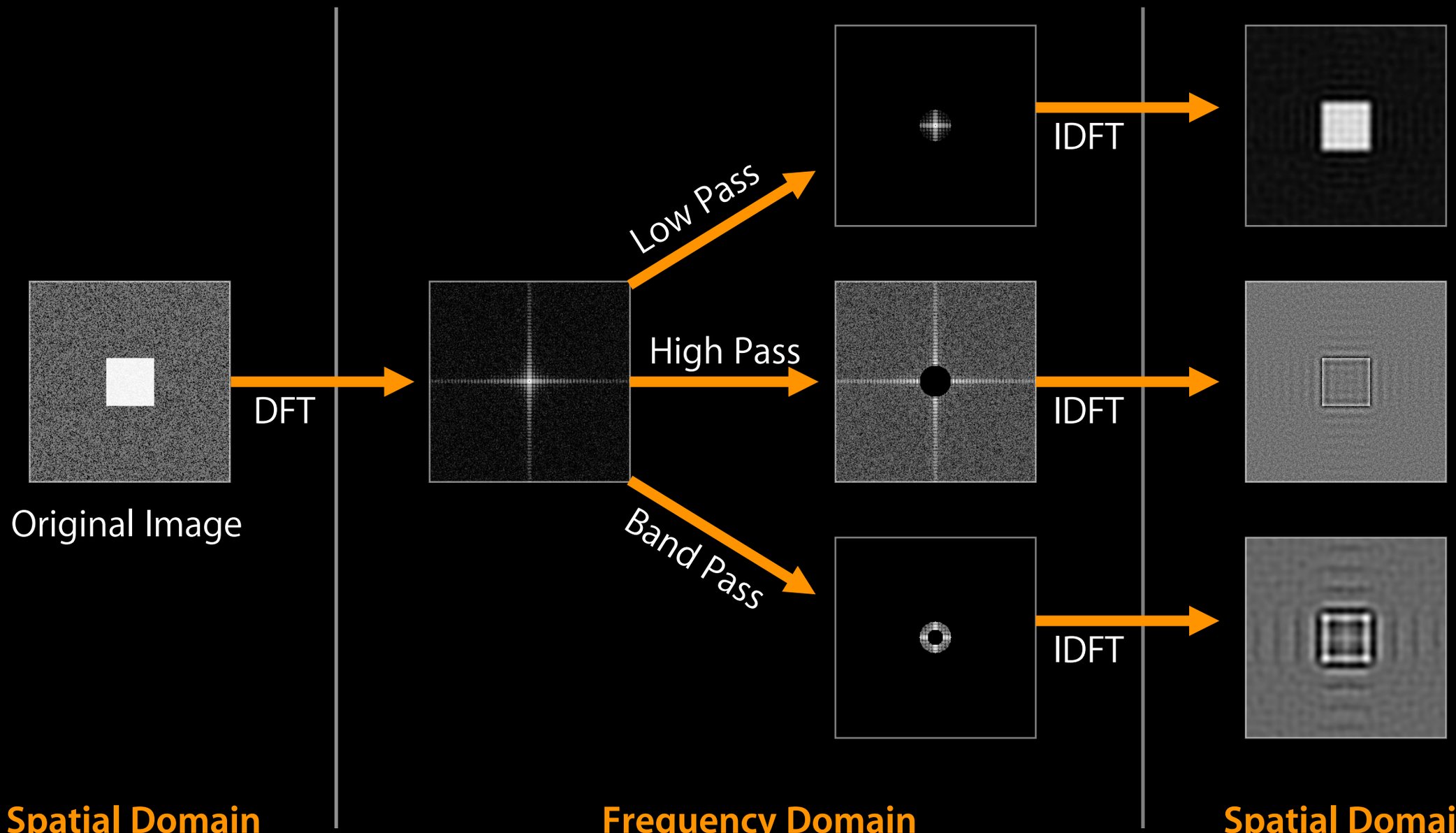
```
!!ARBfp1.0
PARAM  c0 = { 0.111111, 0, 0, 0 };
ATTRIB t0 = fragment.texcoord[0];
TEMP   r0, r1, r2, r3, r8, r9, r10;
OUTPUT oC0 = result.color;
PARAM  c[4] = { program.env[10..13] };

# Compute the tex coords for the taps
ADD r0.xy, t0, c[0];
ADD r1.xy, t0, c[1];
ADD r2.xy, t0, c[2];
ADD r3.xy, t0, c[3];

# sample the texture
TEX r0, r0, texture[0], 2D;
TEX r1, r1, texture[0], 2D;
TEX r2, r2, texture[0], 2D;
TEX r3, r3, texture[0], 2D;

# subtract the variances from each other and test against 0.
# So...if ((r0.a-r1.a) > 0)  then r1 has the lower of the two variances.
ADD r8.w, r0.w,-r1.w;
CMP r8, r8.w, r1, r0;

# subtract again on the next two samples.
ADD r9.w, r2.w,-r3.w;
CMP r9, r9.w, r3, r2;

# Now, take the two winners of the previous two compares and compare one last time.
ADD r10.w, r8.w,-r9.w;
CMP r10, r10.w, r9, r8;

# r10 has the mean of the 3x3 region that has the lowest variance.
MOV oC0, r10;
END
```

FFT Implementation Details

- Cooley and Tukey ***Decimation in Time*** implementation using GPU
  - log2(width) + log2(height) + 2 rendering passes
  - For a 512x512 image, that's 20 rendering passes
  - Ping-pongs between two renderable textures. One render for each step of the transform
  - First does horizontal scramble using "scramble map" to do dependent texture reads from original image
  - Then does log2(width) butterfly passes
  - Vertical scramble using "scramble map" again
  - Finally, does log2(height) butterfly passes

Discrete Fourier Transform

Cooley and Tukey* Decimation in Time FFT Algorithm on GPU



Post-Processing Summary

- Optimized Sepia Tone
- Sobel Edge Detection
- Posterized NPR Rendering
- Real-Time FFT

Shadow Techniques Used In Animusic Demo

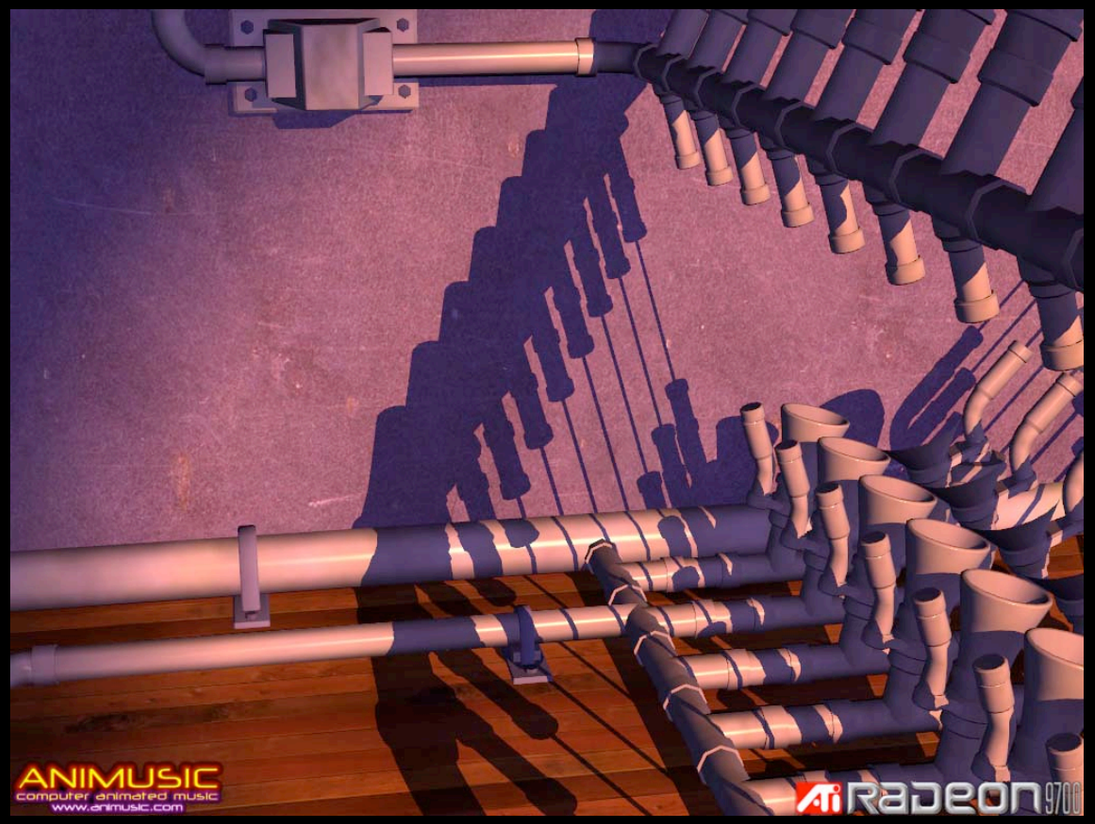
Demo

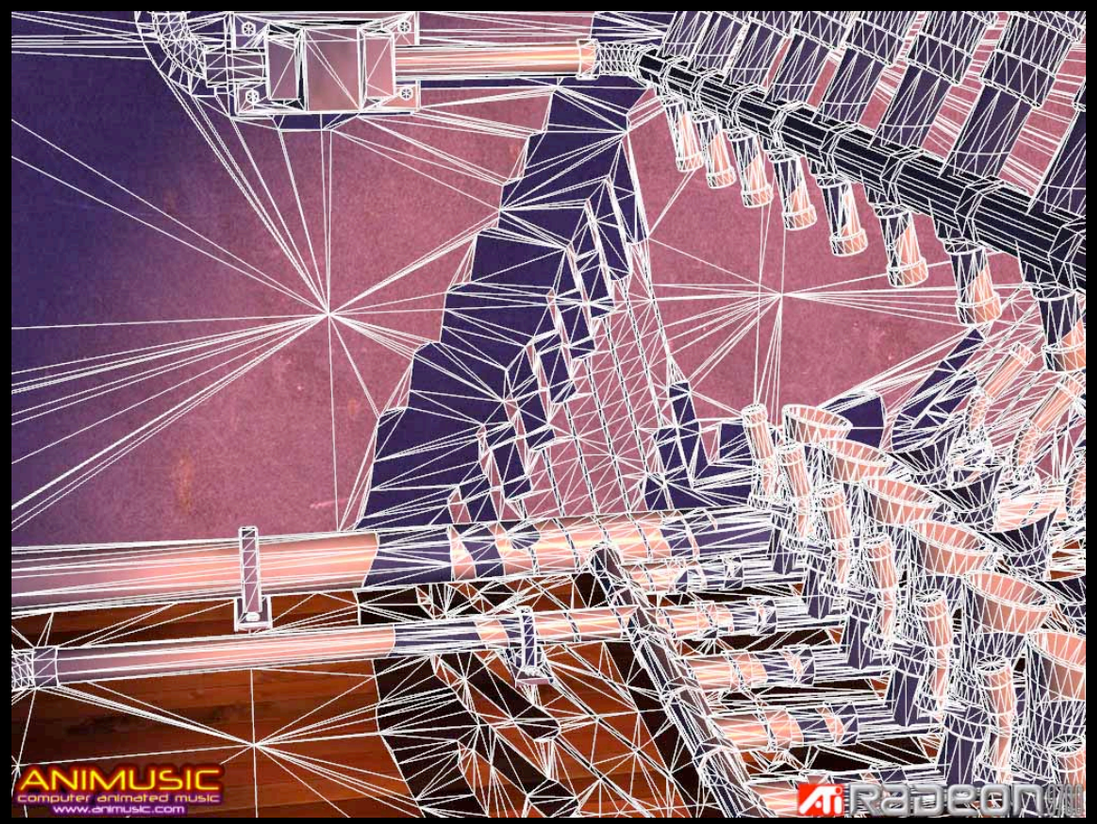
Static Shadows

- Shadows cast by non-moving light sources
- Scene geometry that doesn’t move
  - Terrain, rocks, buildings, etc.
- Great opportunity to optimize out the brute-force nature of dynamic shadow volumes!

Static Shadows in ATI's Demo Engine

- Precomputed shadows are cut directly into the artist-generated geometry



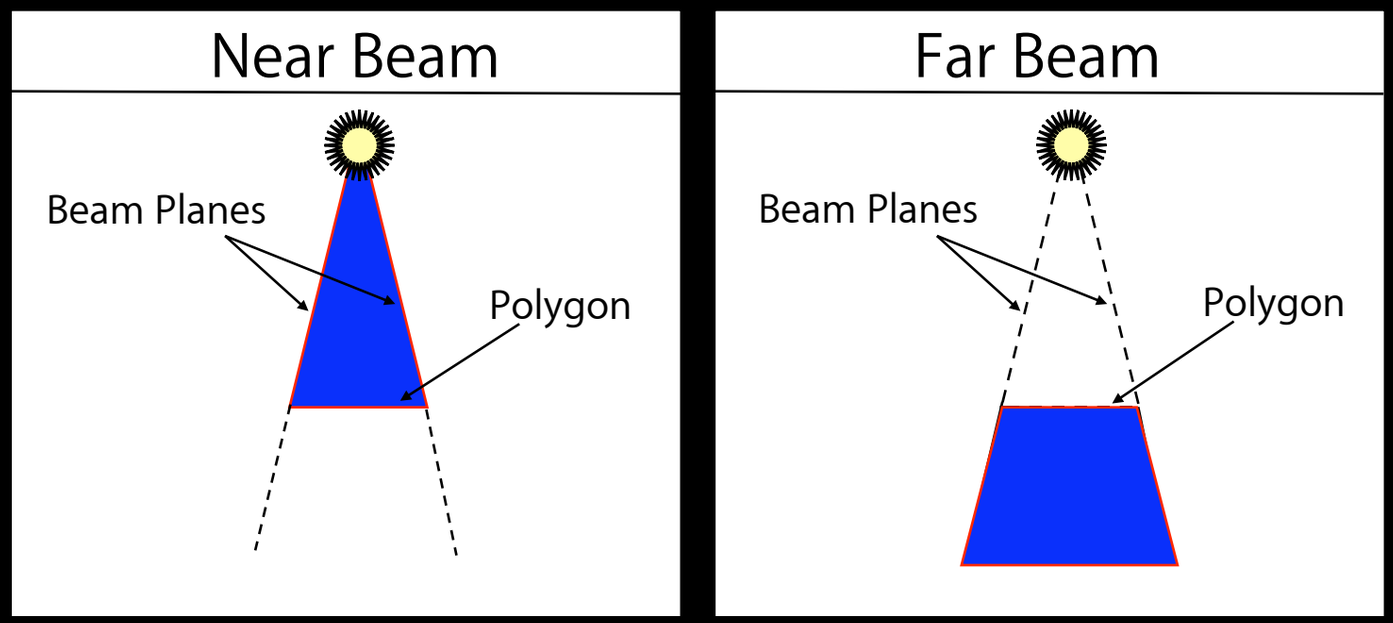


Advantages

- Looks like global stencil shadow volumes without the fill overhead!
- We can draw the polygons that are in shadow with a simpler vertex and pixel shader since fewer lights are affecting those pixels
- The light color that casts the shadows are still animated! Dimming and color change is possible.

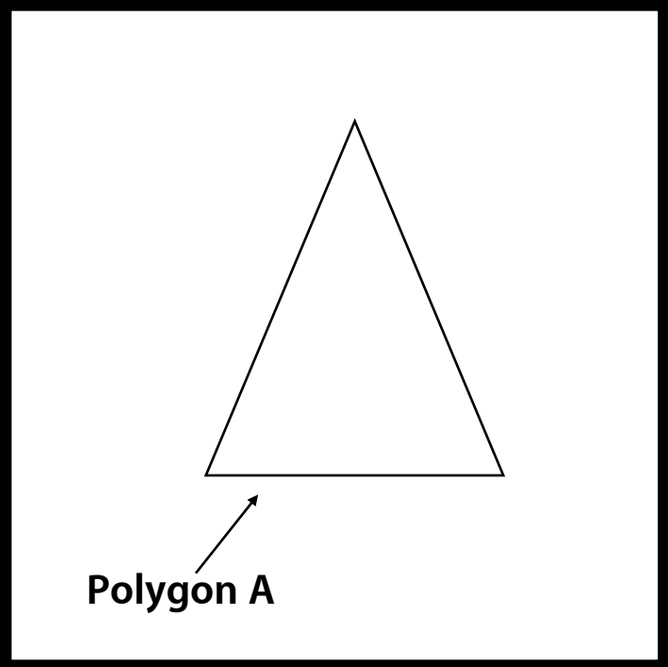
Beam Basics

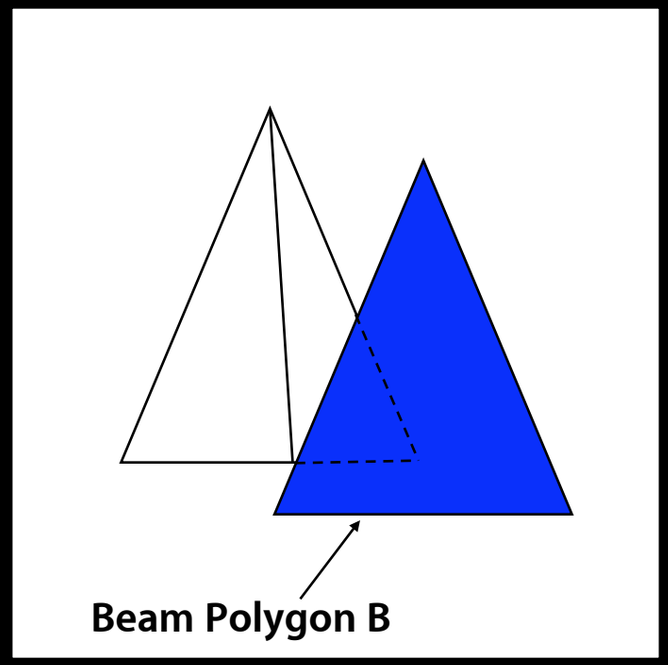
- A beam is a closed volume created by 4 planes
- The 4 planes are constructed from a single triangle and a light's position
- 3 of the 4 planes are defined by an edge of the polygon and the light's position
- The 4th plane is simply the plane of the triangle
- Near vs. Far beam: flip the normal of the 4th plane

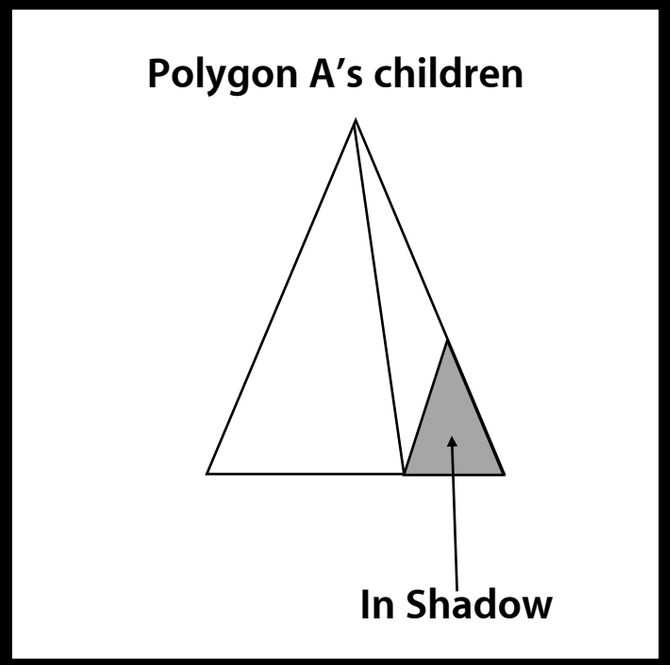


Shadow Cutting Algorithm

- For each polygon in scene (polygon A)
  - For each polygon that falls into polygon A's "near beam" (polygon B)
    - "Far beam" polygon B into polygon A and mark fragments of A that fall inside of beam B as in shadow.

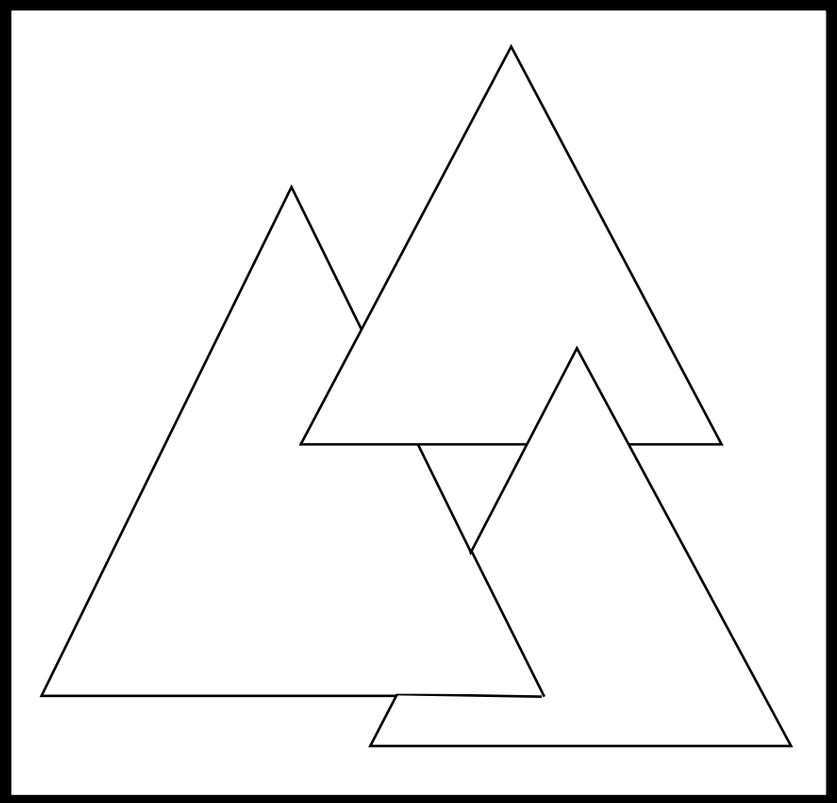


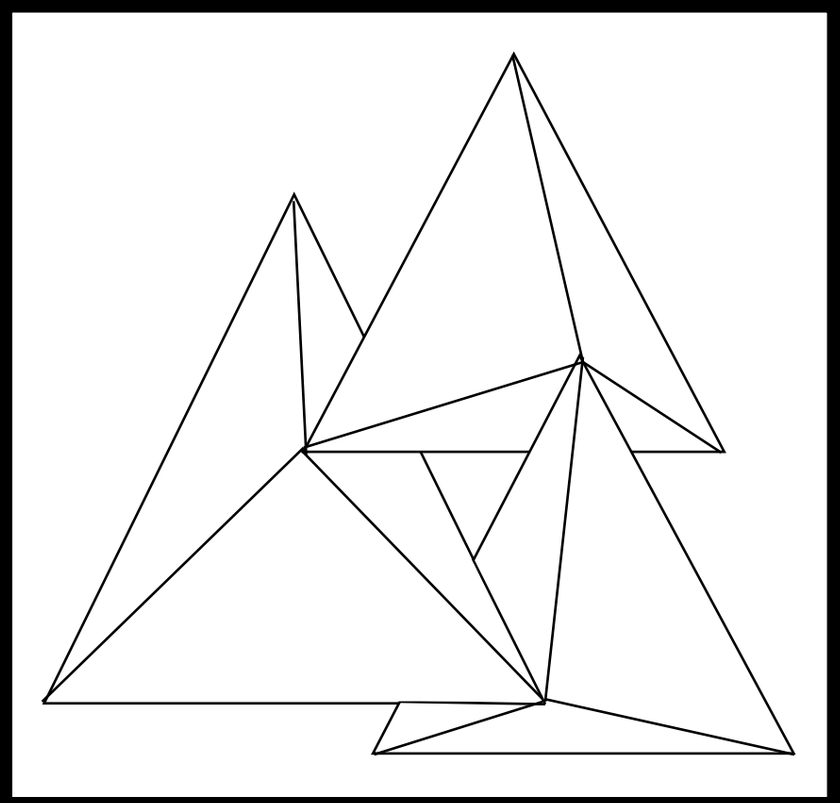




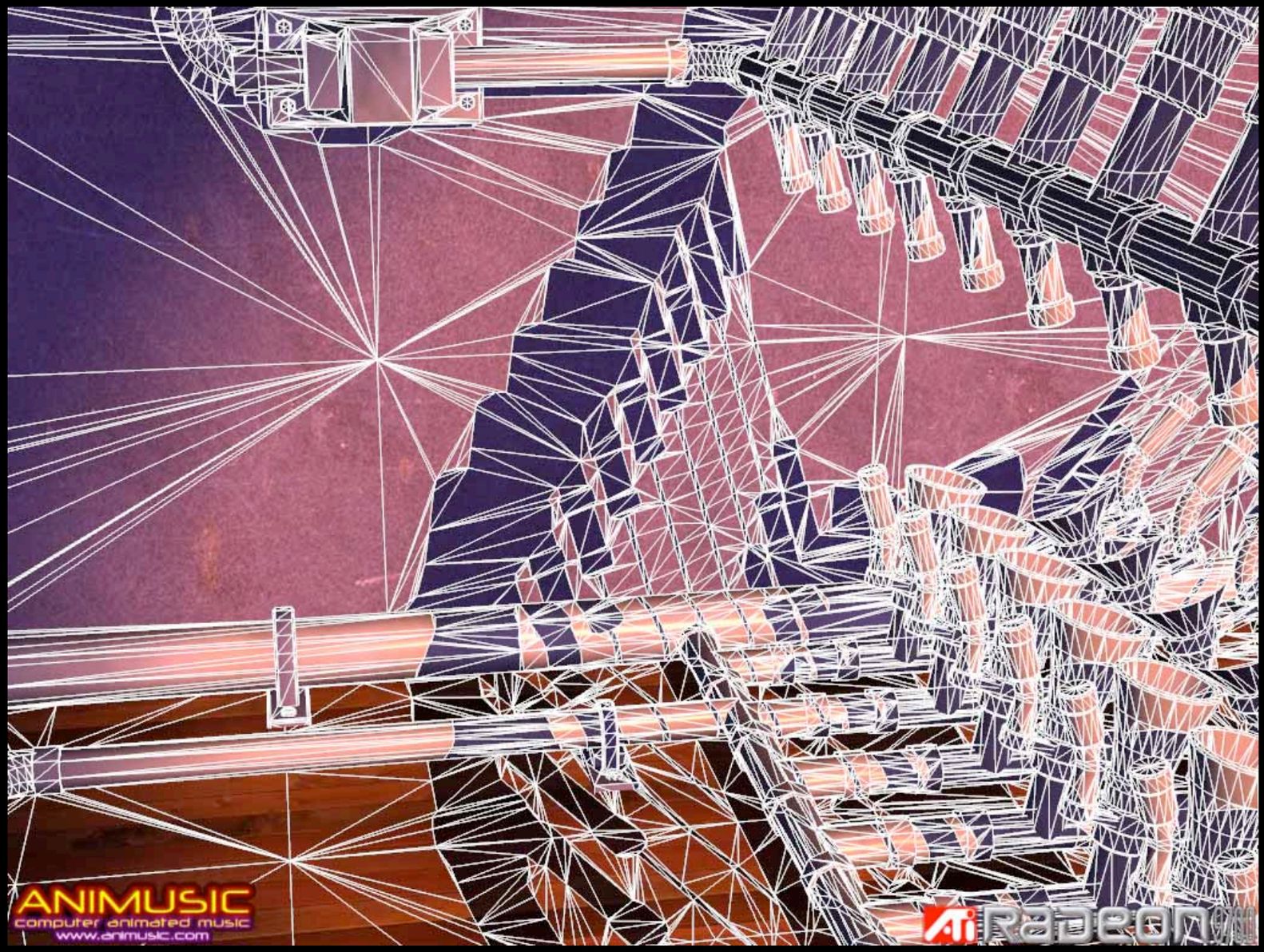
Difficult Cases

- Due to the simplicity of the algorithm, there is no recursion. Only original artist-created polygons form beams.
- This automatically solves for cyclically overlapping polygons without special code:



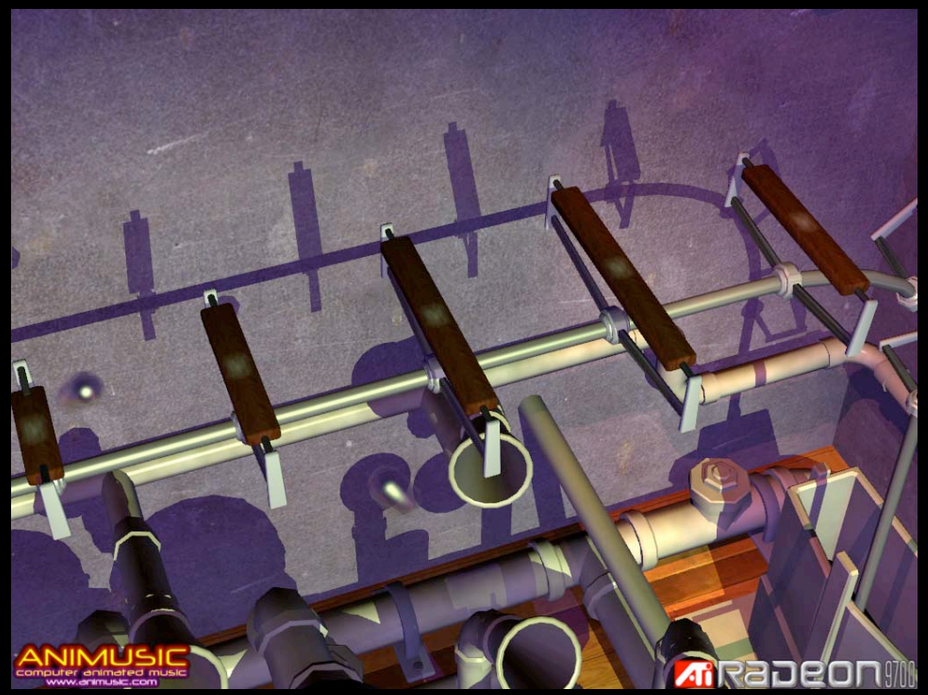


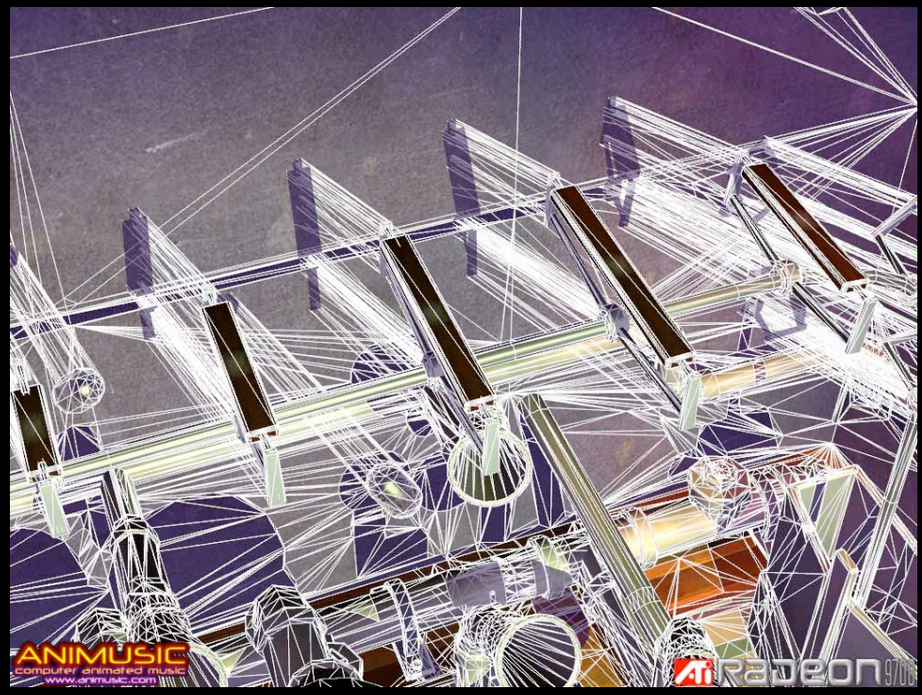
Result of Shadow Cutting



Dynamic Shadows

- Dynamic shadows are used on animated geometry using stencil shadow volumes
- These shadows are combined with the precomputed shadows by drawing a quad over the screen utilizing destination alpha

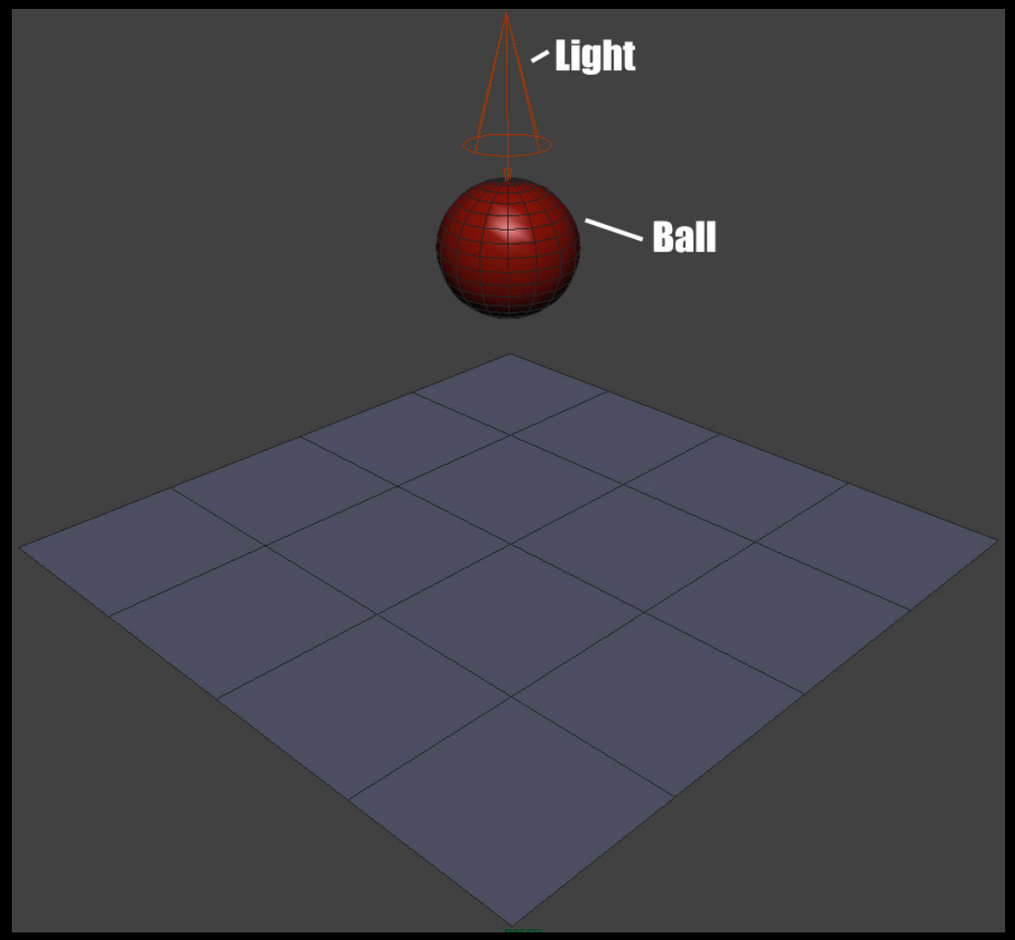


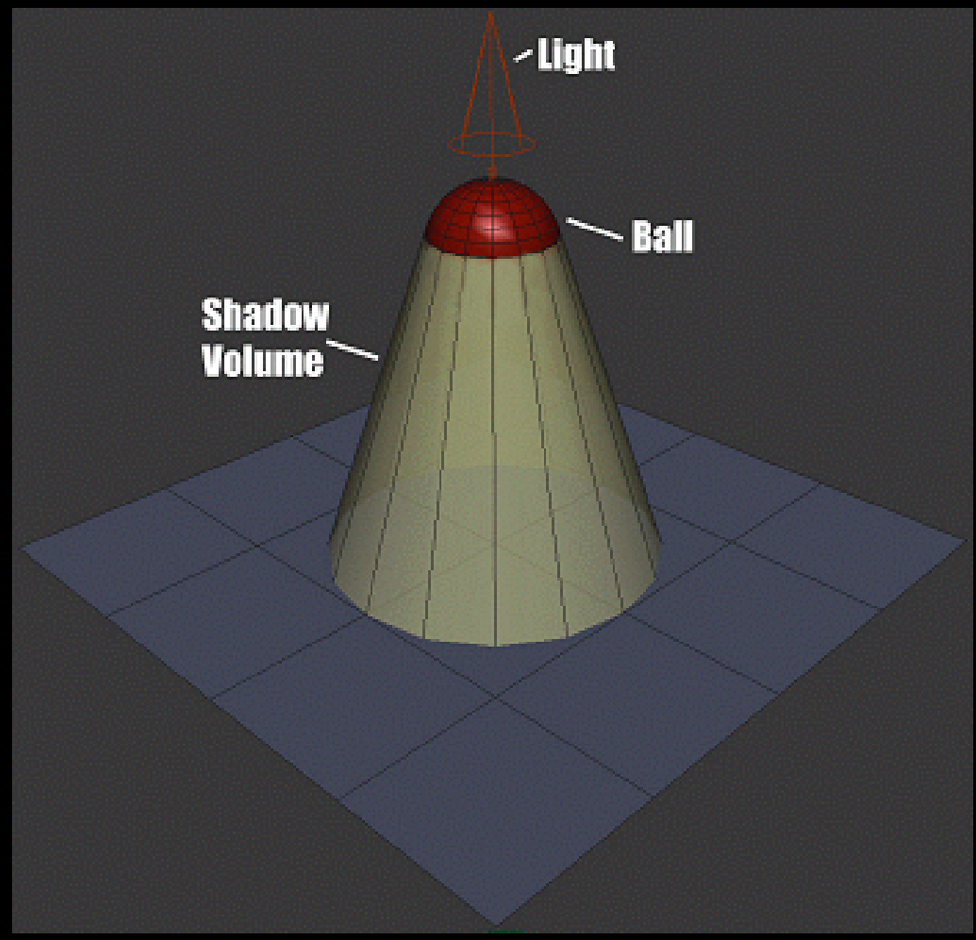


Demo

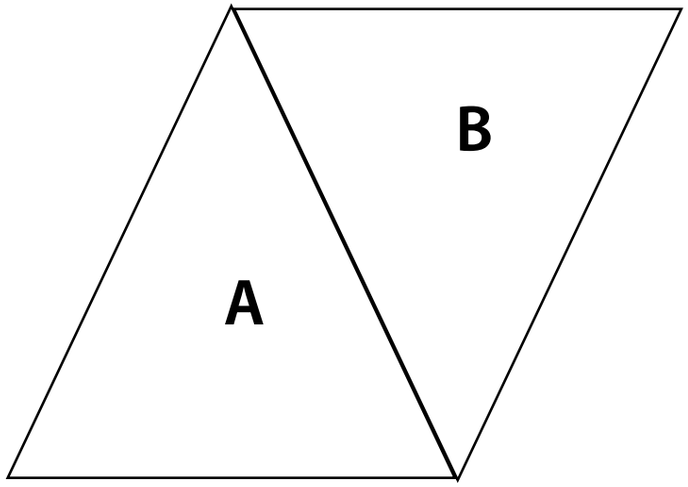
Shadow Volume Basics

- Represent the area that is in shadow
- Any geometry in the volume is in shadow

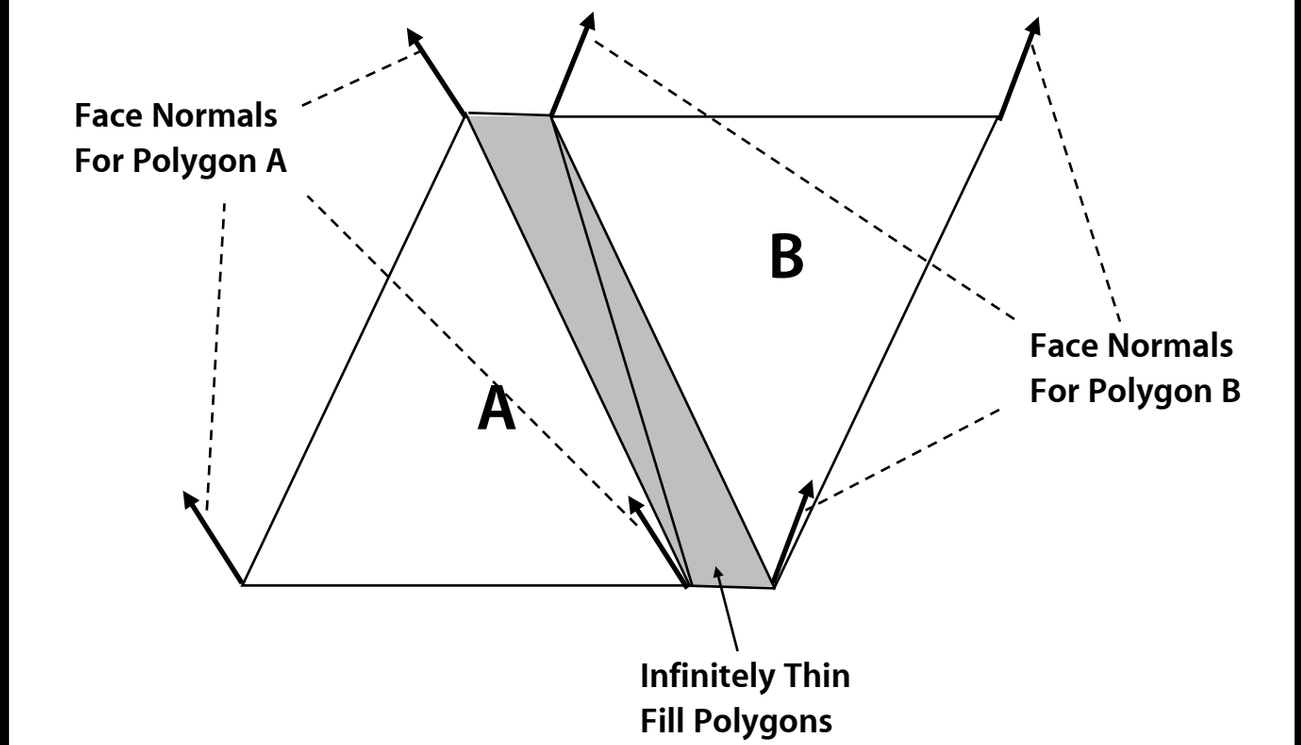




Shadow Volume Extrusion Setup



Original bordering polygons



We insert 2 degenerate polygons between the original polygons which share the appropriate face normal encoded in the vertex

Stencil Buffer

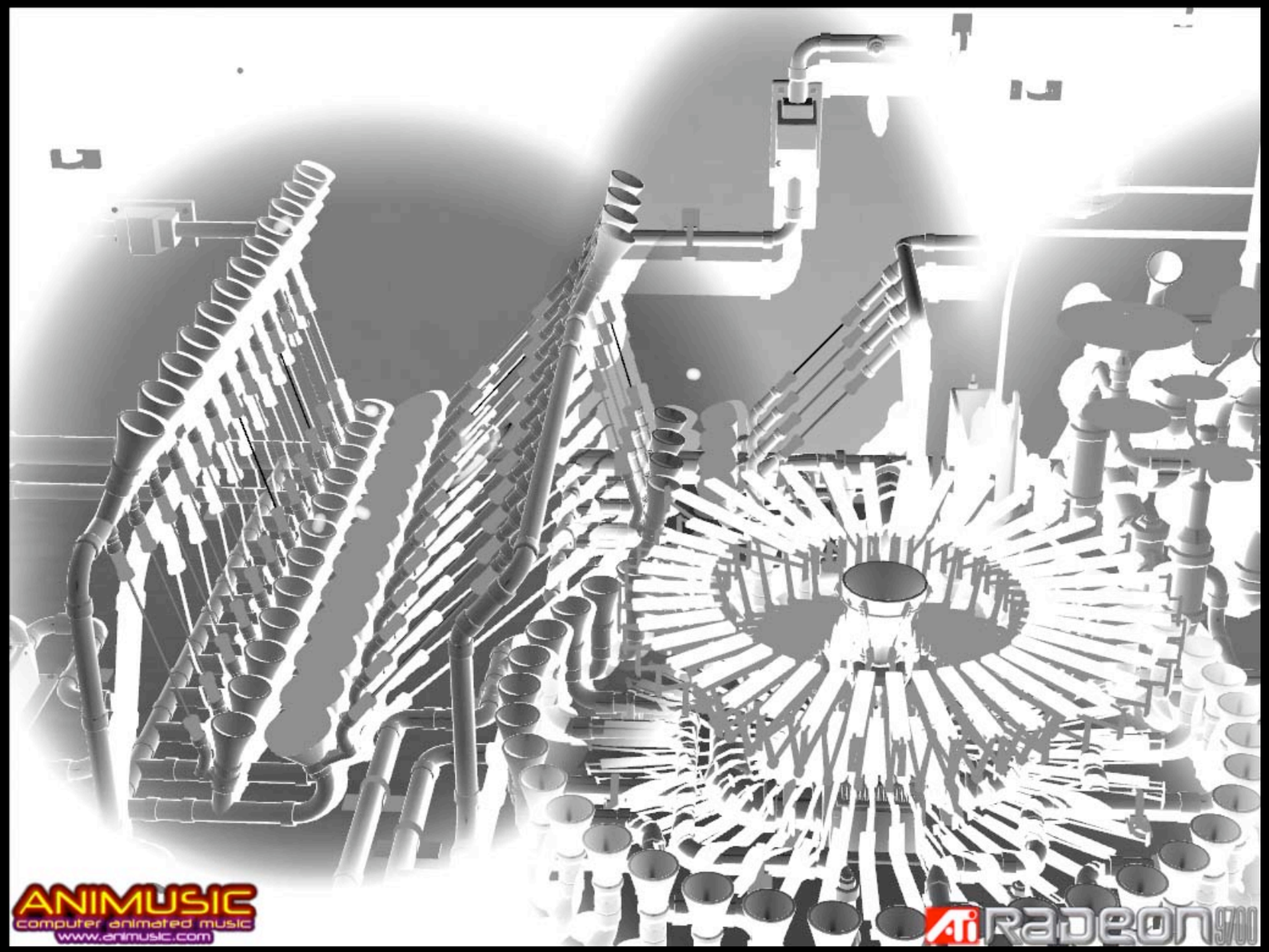
- Dynamic shadow volumes are drawn into the stencil buffer
- Pixels are essentially tagged as in or out of shadow based on their stencil value

- How do we get the shadows into the color buffer?…

Destination Alpha

- As the scene is drawn initially, we also write a value to destination alpha
- The dest alpha value solves the following equation per-pixel:
  - destColor * destAlpha = ambient
  - destAlpha = ambient / destColor
- All pixel shaders write something useful to destination alpha as the scene is being drawn

Contents of Destination Alpha



Demo

Shadow Quad

- A full-screen quad is drawn with the following blending enabled:
  - SrcBlend = Zero
  - DestBlend = DestAlpha

- This provides:
  - SrcColor*Zero + DestColor*DestAlpha

Shadow Summary

- Differences between static and dynamic shadows
- Explained shadow cutting algorithm
- Shadow volume basics
- Dim factor in destination alpha brings it all together

For More Information

References

Documentation

“Real Time Rendering”—Moller & Haines

“Game Programming Gems 4”, edited by Andrew Kirmse, Charles River Media 2004

“ShaderX: Vertex and Pixel Shader Tips and Tricks”, edited by Wolfgang Engel, Wordware 2002

“ShaderX2”, edited by Wolfgang Engel, Wordware 2003

“Digital Image Processing”—Gonzales & Woods

Kuwahara Filter:
http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Smoothin.html

The ATI Normal Mapper tool
http://www.ati.com/developer/tools.html

Who To Contact

Alex Vlachos
avlachos@ati.com

Marwan Y. Ansari
mansari@ati.com

Rav Dhiraj
rdhiraj@ati.com

- Slides will be available on ATI's Developer Relations Web Site
  ati.com/developer
- To be notified about the availability of our tools:
  devrel@ati.com

http://developer.apple.com/wwdc2003/urls.html

Q&A

http://developer.apple.com/wwdc2003/urls.html

WWDC 2003

Worldwide Developers Conference